Edgar H. Sibley
Panel Editor

The state of the art in data compression is arithmetic coding, not the better-known Huffman method. Arithmetic coding gives greater compression, is faster for adaptive models, and clearly separates the model from the channel encoding.

ARITHMETIC CODING FOR DATA COMPRESSION

IAN H. WITTEN, RADFORD M. NEAL, and JOHN G. CLEARY

Arithmetic coding is superior in most respects to the better-known Huffman [10] method. It represents information at least as compactly—sometimes considerably more so. Its performance is optimal without the need for blocking of input data. It encourages a clear separation between the model for representing data and the encoding of information with respect to that model. It accommodates adaptive models easily and is computationally efficient. Yet many authors and practitioners seem unaware of the technique. Indeed there is a widespread belief that Huffman coding cannot be improved upon.

We aim to rectify this situation by presenting an accessible implementation of arithmetic coding and by detailing its performance characteristics. We start by briefly reviewing basic concepts of data compression and introducing the model-based approach that underlies most modern techniques. We then outline the idea of arithmetic coding using a simple example, before presenting programs for both encoding and decoding. In these programs the model occupies a separate module so that different models can easily be used. Next we discuss the construction of fixed and adaptive models and detail the compression efficiency and execution time of the programs, including the effect of different arithmetic word lengths on compression efficiency. Finally, we outline a few applications where arithmetic coding is appropriate.

Financial support for this work has been provided by the Natural Sciences and Engineering Research Council of Canada.

DATA COMPRESSION

To many, data compression conjures up an assortment of ad hoc techniques such as conversion of spaces in text to tabs, creation of special codes for common words, or run-length coding of picture data (e.g., see [8]). This contrasts with the more modern model-based paradigm for coding, where, from an *input string* of symbols and a *model*, an *encoded string* is produced that is (usually) a compressed version of the input. The decoder, which must have access to the same model, regenerates the exact input string from the encoded string. Input symbols are drawn from some well-defined set such as the ASCII or binary alphabets; the encoded string is a plain sequence of bits. The model is a way of calculating, in any given context, the distribution of probabilities for the next input symbol. It must be possible for the decoder to produce exactly the same probability distribution in the same context. Compression is achieved by transmitting the more probable symbols in fewer bits than the less probable ones.

For example, the model may assign a predetermined probability to each symbol in the ASCII alphabet. No context is involved. These probabilities can be determined by counting frequencies in representative samples of text to be transmitted. Such a *fixed* model is communicated in advance to both encoder and decoder, after which it is used for many messages.

Alternatively, the probabilities that an *adaptive* model assigns may change as each symbol is transmitted, based on the symbol frequencies seen so far in the message. There is no need for a representative sample of text, because each message is treated as an independent unit, starting from scratch. The encoder's model changes with each symbol transmitted, and the decoder's changes with each symbol received, in sympathy.

More complex models can provide more accurate probabilistic predictions and hence achieve greater compression. For example, several characters of previous context could condition the next-symbol probability. Such methods have enabled mixed-case English text to be encoded in around 2.2 bits/character with two quite different kinds of model [4, 6]. Techniques that do not separate modeling from coding so distinctly, like that of Ziv and Lempel [23], do not seem to show such great potential for compression, although they may be appropriate when the aim is raw speed rather than compression performance [22].

The effectiveness of any model can be measured by the entropy of the message with respect to it, usually expressed in bits/symbol. Shannon's fundamental theorem of coding states that, given messages randomly generated from a model, it is impossible to encode them into less bits (on average) than the entropy of that model [21].

A message can be coded with respect to a model using either Huffman or arithmetic coding. The former method is frequently advocated as the best possible technique for reducing the encoded data rate. It is not. Given that each symbol in the alphabet must translate into an integral number of bits in the encoding, Huffman coding indeed achieves "minimum redundancy." In other words, it performs optimally if all symbol probabilities are integral powers of ½. But this is not normally the case in practice; indeed, Huffman coding can take up to one extra bit per symbol. The worst case is realized by a source in which one symbol has probability approaching unity. Symbols emanating from such a source convey negligible information on average, but require at least one bit to transmit [7]. Arithmetic coding dispenses with the restriction that each symbol must translate into an integral number of bits, thereby coding more efficiently. It actually achieves the theoretical entropy bound to compression efficiency for any source, including the one just mentioned.

In general, sophisticated models expose the deficiencies of Huffman coding more starkly than simple ones. This is because they more often predict symbols with probabilities close to one, the worst case for Huffman coding. For example, the techniques mentioned above that code English text in 2.2 bits/character both use arithmetic coding as the final step, and performance would be impacted severely if Huffman coding were substituted. Nevertheless, since our topic is coding and not modeling, the illustrations in this article all employ simple models. Even so, as we shall see, Huffman coding is inferior to arithmetic coding.

The basic concept of arithmetic coding can be traced back to Elias in the early 1960s (see [1, pp. 61–62]). Practical techniques were first introduced by Rissanen [16] and Pasco [15], and developed further by Rissanen [17]. Details of the implementation presented here have not appeared in the literature before; Rubin [20] is closest to our approach. The reader interested in the broader class of arithmetic codes is referred to [18]; a tutorial is available in [13]. Despite these publications, the method is not widely known. A number of recent books and papers on data compression mention it only in passing, or not at all.

THE IDEA OF ARITHMETIC CODING

In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows. Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. The more likely symbols reduce the range by less than the unlikely symbols and hence add fewer bits to the message.

Before anything is transmitted, the range for the message is the entire interval [0, 1), denoting the half-open interval $0 \leq x < 1$. As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol. For example, suppose the alphabet is {*a*, *e*, *i*, *o*, *u*, *!*}, and a fixed model is used with probabilities shown in Table I. Imagine transmitting the message *eaii!*. Initially, both encoder and decoder know that the range is [0, 1). After seeing the first symbol, *e*, the encoder narrows it to [0.2, 0.5), the range the model allocates to this symbol. The second symbol, *a*, will narrow this new range to the first one-fifth of it, since *a* has been

TABLE I. Example Fixed Model for Alphabet {*a*, *e*, *i*, *o*, *u*, *!*}

Symbol	Probability	Range
a	.2	[0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

allocated [0, 0.2). This produces [0.2, 0.26), since the previous range was 0.3 units long and one-fifth of that is 0.06. The next symbol, *i*, is allocated [0.5, 0.6), which when applied to [0.2, 0.26) gives the smaller range [0.23, 0.236). Proceeding in this way, the encoded message builds up as follows:

Initially	[0,	1)
After seeing *e*	[0.2,	0.5)
a	[0.2,	0.26)
i	[0.23,	0.236)
i	[0.233,	0.2336)
!	[0.23354,	0.2336)

Figure 1 shows another representation of the encoding process. The vertical bars with ticks represent the symbol probabilities stipulated by the model. After the first symbol has been processed, the model is scaled into the range [0.2, 0.5), as shown in

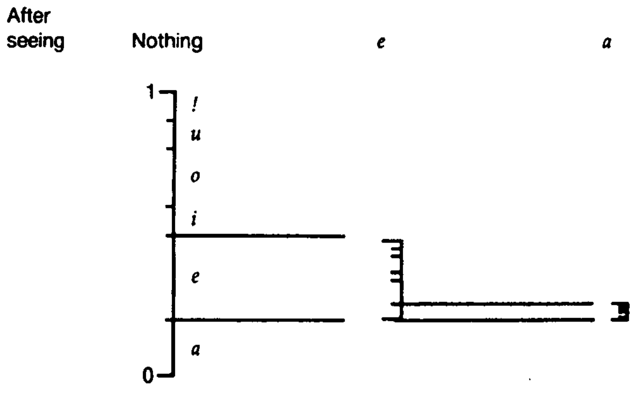

FIGURE 1a. Representation of the Arithmetic Coding Process

Figure 1a. The second symbol scales it again into the range [0.2, 0.26). But the picture cannot be continued in this way without a magnifying glass! Consequently, Figure 1b shows the ranges expanded to full height at every stage and marked with a scale that gives the endpoints as numbers.

Suppose all the decoder knows about the message is the final range, [0.23354, 0.2336). It can immediately deduce that the first character was *e*, since the range lies entirely within the space the model of Table I allocates for *e*. Now it can simulate the operation of the *en*coder:

Initially	[0,	1)
After seeing *e*	[0.2,	0.5)

This makes it clear that the second character is *a*, since this will produce the range

After seeing *a*	[0.2, 0.26),

which entirely encloses the given range [0.23354, 0.2336). Proceeding like this, the decoder can identify the whole message.

It is not really necessary for the decoder to know both ends of the range produced by the encoder. Instead, a single number within the range—for example, 0.23355—will suffice. (Other numbers, like 0.23354, 0.23357, or even 0.23354321, would do just as well.) However, the decoder will face the problem of detecting the end of the message, to determine when to stop decoding. After all, the single number 0.0 could represent any of *a*, *aa*, *aaa*, *aaaa*, ... To resolve the ambiguity, we ensure that each message ends with a special EOF symbol known to both encoder and decoder. For the alphabet of Table I, *!* will be used to terminate messages, and only to termi-

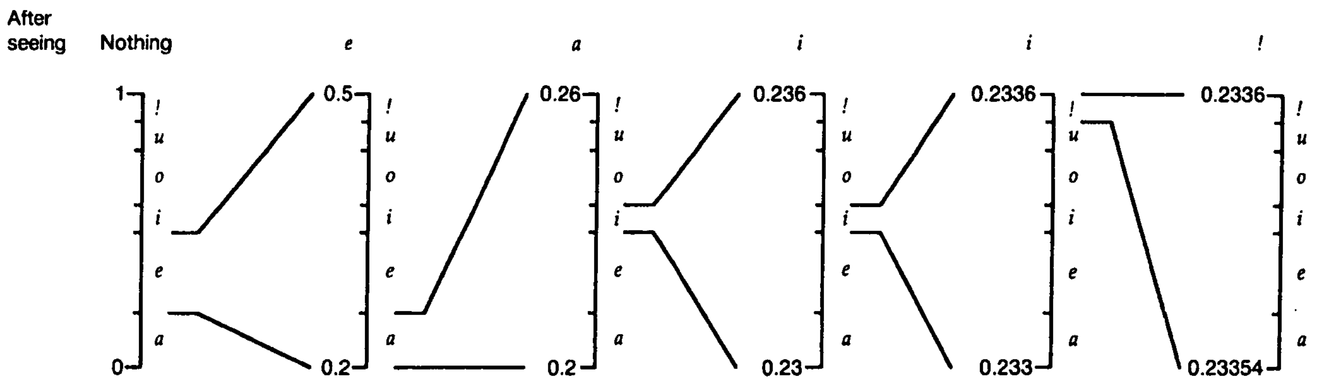

FIGURE 1b. Representation of the Arithmetic Coding Process with the Interval Scaled Up at Each Stage

```
/* ARITHMETIC ENCODING ALGORITHM. */

/* Call encode_symbol repeatedly for each symbol in the message.         */
/* Ensure that a distinguished "terminator" symbol is encoded last, then */
/* transmit any value in the range [low, high).                          */

encode_symbol(symbol, cum_freq)
    range = high - low
    high  = low + range*cum_freq[symbol-1]
    low   = low + range*cum_freq[symbol]


/* ARITHMETIC DECODING ALGORITHM. */

/* "Value" is the number that has been received.                         */
/* Continue calling decode_symbol until the terminator symbol is returned. */

decode_symbol(cum_freq)
    find symbol such that
        cum_freq[symbol] <= (value-low)/(high-low) < cum_freq[symbol-1]
                          /* This ensures that value lies within the new  */
                          /* [low, high) range that will be calculated by */
                          /* the following lines of code.                 */

    range = high - low
    high  = low + range*cum_freq[symbol-1]
    low   = low + range*cum_freq[symbol]
    return symbol
```

FIGURE 2. Pseudocode for the Encoding and Decoding Procedures

nate messages. When the decoder sees this symbol, it stops decoding.

Relative to the fixed model of Table I, the entropy of the five-symbol message *eaii!* is

$$-\log 0.3 - \log 0.2 - \log 0.1 - \log 0.1 - \log 0.1$$
$$= -\log 0.00006 \approx 4.22$$

(using base 10, since the above encoding was performed in decimal). This explains why it takes five decimal digits to encode the message. In fact, the size of the final range is 0.2336 − 0.23354 = 0.00006, and the entropy is the negative logarithm of this figure. Of course, we normally work in binary, transmitting binary digits and measuring entropy in bits.

Five decimal digits seems a lot to encode a message comprising four vowels! It is perhaps unfortunate that our example ended up by expanding rather than compressing. Needless to say, however, different models will give different entropies. The best single-character model of the message *eaii!* is the set of symbol frequencies {*e*(0.2), *a*(0.2), *i*(0.4), *!*(0.2)}, which gives an entropy of 2.89 decimal digits. Using this model the encoding would be only three digits long. Moreover, as noted earlier, more sophisticated models give much better performance in general.

A PROGRAM FOR ARITHMETIC CODING

Figure 2 shows a pseudocode fragment that summarizes the encoding and decoding procedures developed in the last section. Symbols are numbered, 1, 2, 3, ... The frequency range for the *i*th symbol is from *cum_freq*[*i*] to *cum_freq*[*i* − 1]. As *i* decreases, *cum_freq*[*i*] increases, and *cum_freq*[0] = 1. (The reason for this "backwards" convention is that *cum_freq*[0] will later contain a normalizing factor, and it will be convenient to have it begin the array.) The "current interval" is [*low*, *high*), and for both encoding and decoding, this should be initialized to [0, 1).

Unfortunately, Figure 2 is overly simplistic. In practice, there are several factors that complicate both encoding and decoding:

Incremental transmission and reception. The encode algorithm as described does not transmit anything until the entire message has been encoded; neither does the decode algorithm begin decoding until it has received the complete transmission. In most applications an incremental mode of operation is necessary.

The desire to use integer arithmetic. The precision required to represent the [*low*, *high*) interval grows with the length of the message. Incremental operation will help overcome this, but the potential for overflow and underflow must still be examined carefully.

Representing the model so that it can be consulted efficiently. The representation used for the model should minimize the time required for the decode algorithm to identify the next symbol. Moreover, an adaptive model should be organized to minimize the time-consuming task of maintaining cumulative frequencies.

Figure 3 shows working code, in C, for arithmetic encoding and decoding. It is considerably more detailed than the bare-bones sketch of Figure 2! Implementations of two different models are given in Figure 4; the Figure 3 code can use either one.

The remainder of this section examines the code of Figure 3 more closely, and includes a proof that decoding is still correct in the integer implementation and a review of constraints on word lengths in the program.

arithmetic_coding.h

```
/* DECLARATIONS USED FOR ARITHMETIC ENCODING AND DECODING */


/* SIZE OF ARITHMETIC CODE VALUES. */

#define Code_value_bits 16              /* Number of bits in a code value   */
typedef long code_value;                /* Type of an arithmetic code value */

#define Top_value (((long)1<<Code_value_bits)-1)      /* Largest code value */


/* HALF AND QUARTER POINTS IN THE CODE VALUE RANGE. */

#define First_qtr (Top_value/4+1)       /* Point after first quarter        */
#define Half      (2*First_qtr)         /* Point after first half           */
#define Third_qtr (3*First_qtr)         /* Point after third quarter        */
```

model.h

```
/* INTERFACE TO THE MODEL. */


/* THE SET OF SYMBOLS THAT MAY BE ENCODED. */

#define No_of_chars 256                 /* Number of character symbols      */
#define EOF_symbol (No_of_chars+1)      /* Index of EOF symbol              */

#define No_of_symbols (No_of_chars+1)   /* Total number of symbols          */


/* TRANSLATION TABLES BETWEEN CHARACTERS AND SYMBOL INDEXES. */

int char_to_index[No_of_chars];         /* To index from character          */
unsigned char index_to_char[No_of_symbols+1]; /* To character from index    */


/* CUMULATIVE FREQUENCY TABLE. */

#define Max_frequency 16383             /* Maximum allowed frequency count  */
                                        /*   2^14 - 1                       */
int cum_freq[No_of_symbols+1];          /* Cumulative symbol frequencies    */
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding

```
encode.c
```

```
/* MAIN PROGRAM FOR ENCODING. */

#include <stdio.h>
#include "model.h"

main()
{   start_model();                              /* Set up other modules.     */
    start_outputing_bits();
    start_encoding();
    for (;;) {                                  /* Loop through characters. */
        int ch; int symbol;
        ch = getc(stdin);                       /* Read the next character. */
        if (ch==EOF) break;                     /* Exit loop on end-of-file.*/
        symbol = char_to_index[ch];             /* Translate to an index.    */
        encode_symbol(symbol,cum_freq);         /* Encode that symbol.       */
        update_model(symbol);                   /* Update the model.         */
    }
    encode_symbol(EOF_symbol,cum_freq);         /* Encode the EOF symbol.    */
    done_encoding();                            /* Send the last few bits.   */
    done_outputing_bits();
    exit(0);
}
```

```
arithmetic_encode.c
```

```
/* ARITHMETIC ENCODING ALGORITHM. */

#include "arithmetic_coding.h"

static void bit_plus_follow();  /* Routine that follows                      */


/* CURRENT STATE OF THE ENCODING. */

static code_value low, high;    /* Ends of the current code region           */
static long bits_to_follow;     /* Number of opposite bits to output after   */
                                /* the next bit.                             */


/* START ENCODING A STREAM OF SYMBOLS. */

start_encoding()
{   low = 0;                                    /* Full code range.          */
    high = Top_value;
    bits_to_follow = 0;                         /* No bits to follow next.   */
}


/* ENCODE A SYMBOL. */

encode_symbol(symbol,cum_freq)
    int symbol;                 /* Symbol to encode                          */
    int cum_freq[];             /* Cumulative symbol frequencies             */
{   long range;                 /* Size of the current code region           */
    range = (long)(high-low)+1;
    high = low +                                /* Narrow the code region    */
      (range*cum_freq[symbol-1])/cum_freq[0]-1; /* to that allotted to this */
    low = low +                                 /* symbol.                   */
      (range*cum_freq[symbol])/cum_freq[0];
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding (*continued*)

```
    for (;;) {                                  /* Loop to output bits.     */
        if (high<Half) {
            bit_plus_follow(0);                 /* Output 0 if in low half. */
        }
        else if (low>=Half) {                   /* Output 1 if in high half.*/
            bit_plus_follow(1);
            low -= Half;
            high -= Half;                       /* Subtract offset to top.  */
        }
        else if (low>=First_qtr                 /* Output an opposite bit   */
              && high<Third_qtr) {              /* later if in middle half. */
            bits_to_follow += 1;
            low -= First_qtr;                   /* Subtract offset to middle*/
            high -= First_qtr;
        }
        else break;                             /* Otherwise exit loop.     */
        low = 2*low;
        high = 2*high+1;                        /* Scale up code range.     */
    }
}


/* FINISH ENCODING THE STREAM. */

done_encoding()
{   bits_to_follow += 1;                        /* Output two bits that     */
    if (low<First_qtr) bit_plus_follow(0);      /* select the quarter that  */
    else bit_plus_follow(1);                    /* the current code range   */
}                                               /* contains.                */


/* OUTPUT BITS PLUS FOLLOWING OPPOSITE BITS. */

static void bit_plus_follow(bit)
    int bit;
{   output_bit(bit);                            /* Output the bit.          */
    while (bits_to_follow>0) {
        output_bit(!bit);                       /* Output bits_to_follow    */
        bits_to_follow -= 1;                    /* opposite bits. Set       */
    }                                           /* bits_to_follow to zero.  */
}
```

decode.c

```
/* MAIN PROGRAM FOR DECODING. */

#include <stdio.h>
#include "model.h"

main()
{   start_model();                              /* Set up other modules.    */
    start_inputing_bits();
    start_decoding();
    for (;;) {                                  /* Loop through characters. */
        int ch; int symbol;
        symbol = decode_symbol(cum_freq);       /* Decode next symbol.      */
        if (symbol==EOF_symbol) break;          /* Exit loop if EOF symbol. */
        ch = index_to_char[symbol];             /* Translate to a character.*/
        putc(ch,stdout);                        /* Write that character.    */
        update_model(symbol);                   /* Update the model.        */
    }
    exit(0);
}
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding (*continued*)

arithmetic_decode.c

```
/* ARITHMETIC DECODING ALGORITHM. */

#include "arithmetic_coding.h"


/* CURRENT STATE OF THE DECODING. */

static code_value value;        /* Currently-seen code value              */
static code_value low, high;    /* Ends of current code region            */


/* START DECODING A STREAM OF SYMBOLS. */

start_decoding()
{   int i;
    value = 0;                                    /* Input bits to fill the   */
    for (i = 1; i<=Code_value_bits; i++) {        /* code value.              */
        value = 2*value+input_bit();
    }
    low = 0;                                      /* Full code range.         */
    high = Top_value;
}


/* DECODE THE NEXT SYMBOL. */

int decode_symbol(cum_freq)
    int cum_freq[];             /* Cumulative symbol frequencies          */
{   long range;                 /* Size of current code region            */
    int cum;                    /* Cumulative frequency calculated        */
    int symbol;                 /* Symbol decoded                         */
    range = (long)(high-low)+1;
    cum =                                         /* Find cum freq for value. */
      (((long)(value-low)+1)*cum_freq[0]-1)/range;
    for (symbol = 1; cum_freq[symbol]>cum; symbol++) ; /* Then find symbol. */
    high = low +                                  /* Narrow the code region   */
      (range*cum_freq[symbol-1])/cum_freq[0]-1;   /* to that allotted to this */
    low = low +                                   /* symbol.                  */
      (range*cum_freq[symbol])/cum_freq[0];
    for (;;) {                                    /* Loop to get rid of bits. */
        if (high<Half) {
            /* nothing */                         /* Expand low half.         */
        }
        else if (low>=Half) {                     /* Expand high half.        */
            value -= Half;
            low -= Half;                          /* Subtract offset to top.  */
            high -= Half;
        }
        else if (low>=First_qtr                   /* Expand middle half.      */
              && high<Third_qtr) {
            value -= First_qtr;
            low -= First_qtr;                     /* Subtract offset to middle*/
            high -= First_qtr;
        }
        else break;                               /* Otherwise exit loop.     */
        low = 2*low;
        high = 2*high+1;                          /* Scale up code range.     */
        value = 2*value+input_bit();              /* Move in next input bit.  */
    }
    return symbol;
}
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding (*continued*)

bit_input.c

```
/* BIT INPUT ROUTINES. */

#include <stdio.h>
#include "arithmetic_coding.h"


/* THE BIT BUFFER. */

static int buffer;              /* Bits waiting to be input              */
static int bits_to_go;          /* Number of bits still in buffer        */
static int garbage_bits;        /* Number of bits past end-of-file       */


/* INITIALIZE BIT INPUT. */

start_inputing_bits()
{   bits_to_go = 0;                             /* Buffer starts out with   */
    garbage_bits = 0;                           /* no bits in it.           */
}


/* INPUT A BIT. */

int input_bit()
{   int t;
    if (bits_to_go==0) {                        /* Read the next byte if no */
        buffer = getc(stdin);                   /* bits are left in buffer. */
        if (buffer==EOF) {
            garbage_bits += 1;                      /* Return arbitrary bits*/
            if (garbage_bits>Code_value_bits-2) {   /* after eof, but check */
                fprintf(stderr,"Bad input file\n"); /* for too many such.   */
                exit(-1);
            }
        }
        bits_to_go = 8;
    }
    t = buffer&1;                               /* Return the next bit from */
    buffer >>= 1;                               /* the bottom of the byte.  */
    bits_to_go -= 1;
    return t;
}
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding (*continued*)

bit_output.c

```
/* BIT OUTPUT ROUTINES. */

#include <stdio.h>


/* THE BIT BUFFER. */

static int buffer;              /* Bits buffered for output              */
static int bits_to_go;          /* Number of bits free in buffer         */


/* INITIALIZE FOR BIT OUTPUT. */

start_outputing_bits()
{   buffer = 0;                                  /* Buffer is empty to start */
    bits_to_go= 8;                               /* with.                    */
}


/* OUTPUT A BIT. */

output_bit(bit)
    int bit;
{   buffer >>= 1;                                /* Put bit in top of buffer.*/
    if (bit) buffer |= 0x80;
    bits_to_go -= 1;
    if (bits_to_go==0) {                         /* Output buffer if it is   */
        putc(buffer,stdout);                     /* now full.                */
        bits_to_go = 8;
    }
}


/* FLUSH OUT THE LAST BITS. */

done_outputing_bits()
{   putc(buffer>>bits_to_go,stdout);
}
```

FIGURE 3. C Implementation of Arithmetic Encoding and Decoding (*continued*)

fixed_model.c

```
/* THE FIXED SOURCE MODEL */

#include "model.h"

int freq[No_of_symbols+1] = {
   0,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1, 124,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,

/*       !    "    #    $    %    &    '    (    )    *    +    ,    -    .    / */
1236,    1,  21,   9,   3,   1,  25,  15,   2,   2,   2,   1,  79,  19,  60,   1,

/* 0     1    2    3    4    5    6    7    8    9    :    ;    <    =    >    ? */
  15,   15,   8,   5,   4,   7,   5,   4,   4,   6,   3,   2,   1,   1,   1,   1,

/* @     A    B    C    D    E    F    G    H    I    J    K    L    M    N    O */
   1,   24,  15,  22,  12,  15,  10,   9,  16,  16,   8,   6,  12,  23,  13,  11,

/* P     Q    R    S    T    U    V    W    X    Y    Z    [    \    ]    ^    _ */
  14,    1,  14,  28,  29,   6,   3,  11,   1,   3,   1,   1,   1,   1,   1,   3,

/* '     a    b    c    d    e    f    g    h    i    j    k    l    m    n    o */
   1,  491,  85, 173, 232, 744, 127, 110, 293, 418,   6,  39, 250, 139, 429, 446,

/* p     q    r    s    t    u    v    w    x    y    z    {    |    }    ~      */
 111,    5, 388, 375, 531, 152,  57,  97,  12, 101,   5,   2,   1,   2,   3,   1,

   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1
};


/* INITIALIZE THE MODEL. */

start_model()
{   int i;
    for (i = 0; i<No_of_chars; i++) {           /* Set up tables that       */
        char_to_index[i] = i+1;                 /* translate between symbol */
        index_to_char[i+1] = i;                 /* indexes and characters.  */
    }
    cum_freq[No_of_symbols] = 0;
    for (i = No_of_symbols; i>0; i--) {         /* Set up cumulative        */
        cum_freq[i-1] = cum_freq[i] + freq[i];  /* frequency counts.        */
    }
    if (cum_freq[0] > Max_frequency) abort();   /* Check counts within limit*/
}


/* UPDATE THE MODEL TO ACCOUNT FOR A NEW SYMBOL. */

update_model(symbol)
    int symbol;
{                                               /* Do nothing. */
}
```

FIGURE 4. Fixed and Adaptive Models for Use with Figure 3

adaptive_model.c

```
/* THE ADAPTIVE SOURCE MODEL */

#include "model.h"

int freq[No_of_symbols+1];      /* Symbol frequencies                       */


/* INITIALIZE THE MODEL. */

start_model()
{   int i;
    for (i = 0; i<No_of_chars; i++) {          /* Set up tables that       */
        char_to_index[i] = i+1;                 /* translate between symbol */
        index_to_char[i+1] = i;                 /* indexes and characters.  */
    }
    for (i = 0; i<=No_of_symbols; i++) {        /* Set up initial frequency */
        freq[i] = 1;                            /* counts to be one for all */
        cum_freq[i] = No_of_symbols-i;          /* symbols.                 */
    }
    freq[0] = 0;                                /* Freq[0] must not be the  */
}                                               /* same as freq[1].         */


/* UPDATE THE MODEL TO ACCOUNT FOR A NEW SYMBOL. */

update_model(symbol)
    int symbol;                 /* Index of new symbol                      */
{   int i;                      /* New index for symbol                     */
    if (cum_freq[0]==Max_frequency) {           /* See if frequency counts  */
        int cum;                                /* are at their maximum.    */
        cum = 0;
        for (i = No_of_symbols; i>=0; i--) {    /* If so, halve all the     */
            freq[i] = (freq[i]+1)/2;            /* counts (keeping them     */
            cum_freq[i] = cum;                  /* non-zero).               */
            cum += freq[i];
        }
    }
    for (i = symbol; freq[i]==freq[i-1]; i--) ; /* Find symbol's new index. */
    if (i<symbol) {
        int ch_i, ch_symbol;
        ch_i = index_to_char[i];                /* Update the translation   */
        ch_symbol = index_to_char[symbol];      /* tables if the symbol has */
        index_to_char[i] = ch_symbol;           /* moved.                   */
        index_to_char[symbol] = ch_i;
        char_to_index[ch_i] = symbol;
        char_to_index[ch_symbol] = i;
    }
    freq[i] += 1;                               /* Increment the frequency  */
    while (i>0) {                               /* count for the symbol and */
        i -= 1;                                 /* update the cumulative    */
        cum_freq[i] += 1;                       /* frequencies.             */
    }
}
```

FIGURE 4. Fixed and Adaptive Models for Use with Figure 3 (*continued*)

Representing the Model

Implementations of models are discussed in the next section; here we are concerned only with the interface to the model (lines 20–38). In C, a byte is represented as an integer between 0 and 255 (a *char*). Internally, we represent a byte as an integer between 1 and 257 inclusive (an *index*), EOF being treated as a 257th symbol. It is advantageous to sort the model into frequency order, so as to minimize the number of executions of the decoding loop (line 189). To permit such reordering, the *char/index* translation is implemented as a pair of tables, *index_to_char*[ ] and *char_to_index*[ ]. In one of our models, these tables simply form the *index* by adding 1 to the *char*, but another implements a more complex translation that assigns small indexes to frequently used symbols.

The probabilities in the model are represented as integer frequency counts, and cumulative counts are stored in the array *cum_freq*[ ]. As previously, this array is "backwards," and the total frequency count, which is used to normalize all frequencies, appears in *cum_freq*[0]. Cumulative counts must not exceed a predetermined maximum, *Max_frequency*, and the model implementation must prevent overflow by scaling appropriately. It must also ensure that neighboring values in the *cum_freq*[ ] array differ by at least 1; otherwise the affected symbol cannot be transmitted.

Incremental Transmission and Reception

Unlike Figure 2 the program in Figure 3 represents *low* and *high* as integers. A special data type, *code_value*, is defined for these quantities, together with some useful constants: *Top_value*, representing the largest possible *code_value*, and *First_qtr*, *Half*, and *Third_qtr*, representing parts of the range (lines 6–16). Whereas in Figure 2 the current interval is represented by [*low*, *high*), in Figure 3 it is [*low*, *high*]; that is, the range now includes the value of *high*. Actually, it is more accurate (though more confusing) to say that, in the program in Figure 3, the interval represented is $[low, high + 0.11111 \cdots)$. This is because when the bounds are scaled up to increase the precision, zeros are shifted into the low-order bits of *low*, but ones are shifted into *high*. Although it is possible to write the program to use a different convention, this one has some advantages in simplifying the code.

As the code range narrows, the top bits of *low* and *high* become the same. Any bits that are the same can be transmitted immediately, since they cannot be affected by future narrowing. For encoding, since we know that $low \leq high$, this requires code like

```
for (;;) {
  if (high < Half) {
    output_bit(0);
    low  = 2*low;
    high = 2*high+1;
  }
  else if (low ≥ Half) {
    output_bit(1);
    low = 2*(low-Half);
    high = 2*(high-Half)+1;
  }
  else break;
}
```

which ensures that, upon completion, $low < Half \leq high$. This can be found in lines 95–113 of *encode_symbol*( ), although there are some extra complications caused by underflow possibilities (see the next subsection). Care is taken to shift ones in at the bottom when *high* is scaled, as noted above.

Incremental reception is done using a number called *value* as in Figure 2, in which processed bits flow out the top (high-significance) end and newly received ones flow in the bottom. Initially, *start_decoding*( ) (lines 168–176) fills *value* with received bits. Once *decode_symbol*( ) has identified the next input symbol, it shifts out now-useless high-order bits that are the same in *low* and *high*, shifting *value* by the same amount (and replacing lost bits by fresh input bits at the bottom end):

```
for (;;) {
  if (high < Half) {
    value = 2*value+input_bit( );
    low   = 2*low;
    high  = 2*high+1;
  }
  else if (low > Half) {
    value = 2*(value-Half)+input_bit( );
    low   = 2*(low-Half);
    high  = 2*(high-Half)+1;
  }
  else break;
}
```

(see lines 194–213, again complicated by precautions against underflow, as discussed below).

Proof of Decoding Correctness

At this point it is worth checking that identification of the next symbol by *decode_symbol*( ) works properly. Recall from Figure 2 that *decode_symbol*( ) must use *value* to find the symbol that, when encoded, reduces the range to one that still includes *value*. Lines 186–188 in *decode_symbol*( ) identify the symbol for which

$$cum\_freq[symbol] \le \left\lfloor \frac{(value - low + 1) * cum\_freq[0] - 1}{high - low + 1} \right\rfloor < cum\_freq[symbol - 1],$$

where $\lfloor\,\rfloor$ denotes the "integer part of" function that comes from integer division with truncation. It is shown in the Appendix that this implies

$$low + \left\lfloor \frac{(high - low + 1) * cum\_freq[symbol]}{cum\_freq[0]} \right\rfloor \le v \le low + \left\lfloor \frac{(high - low + 1) * cum\_freq[symbol - 1]}{cum\_freq[0]} \right\rfloor - 1,$$

so that *value* lies within the new interval that *decode_symbol*() calculates in lines 190–193. This is sufficient to guarantee that the decoding operation identifies each symbol correctly.

Underflow

As Figure 1 shows, arithmetic coding works by scaling the cumulative probabilities given by the model into the interval [*low*, *high*] for each character transmitted. Suppose *low* and *high* are very close together—so close that this scaling operation maps some different symbols of the model onto the same integer in the [*low*, *high*] interval. This would be disastrous, because if such a symbol actually occurred it would not be possible to continue encoding. Consequently, the encoder must guarantee that the interval [*low*, *high*] is always large enough to prevent this. The simplest way to do this is to ensure that this interval is at least as large as *Max_frequency*, the maximum allowed cumulative frequency count (line 36).

How could this condition be violated? The bit-shifting operation explained above ensures that *low* and *high* can only become close together when they straddle *Half*. Suppose in fact they become as close as

$$First\_qtr \le low < Half \le high < Third\_qtr.$$

Then the next two bits sent will have opposite polarity, either 01 or 10. For example, if the next bit turns out to be zero (i.e., *high* descends below *Half* and [0, *Half*] is expanded to the full interval), the bit after that will be one, since the range has to be above the midpoint of the expanded interval. Conversely, if the next bit happens to be one, the one after that will be zero. Therefore the interval can safely be expanded right now, if only we remember that, whatever bit actually comes next, its opposite must be transmitted afterwards as well. Thus lines 104–109 expand [*First_qtr*, *Third_qtr*] into the whole interval, remembering in *bits_to_follow* that the bit that is output next must be followed by an opposite bit. This explains why all output is done via *bit_plus_follow*() (lines 128–135), instead of directly with *output_bit*().

But what if, after this operation, it is *still* true that

$$First\_qtr \le low < Half \le high < Third\_qtr?$$

Figure 5 illustrates this situation, where the current [*low*, *high*] range (shown as a thick line) has been expanded a total of three times. Suppose the next bit turns out to be zero, as indicated by the arrow in Figure 5a being below the halfway point. Then the next three bits will be ones, since the arrow is not only in the top half of the bottom half of the original range, but in the top quarter, and moreover the top eighth, of that half—this is why the expansion can occur three times. Similarly, as Figure 5b shows, if the next bit turns out to be a one, it will be followed by three zeros. Consequently, we need only count the number of expansions and follow the next bit by that number of opposites (lines 106 and 131–134).

Using this technique the encoder can guarantee that, after the shifting operations, either

$$low < First\_qtr < Half \le high \qquad (1a)$$

or

$$low < Half < Third\_qtr \le high. \qquad (1b)$$

Therefore, as long as the integer range spanned by the cumulative frequencies fits into a quarter of that provided by *code_values*, the underflow problem cannot occur. This corresponds to the condition

$$Max\_frequency \le \frac{Top\_value + 1}{4} + 1,$$

which is satisfied by Figure 3, since *Max_frequency* $= 2^{14} - 1$ and *Top_value* $= 2^{16} - 1$ (lines 36, 9). More than 14 bits cannot be used to represent cumulative frequency counts without increasing the number of bits allocated to *code_values*.

We have discussed underflow in the encoder only. Since the decoder's job, once each symbol has been decoded, is to track the operation of the encoder, underflow will be avoided if it performs the same expansion operation under the same conditions.

Overflow

Now consider the possibility of overflow in the integer multiplications corresponding to those of Figure 2, which occur in lines 91–94 and 190–193

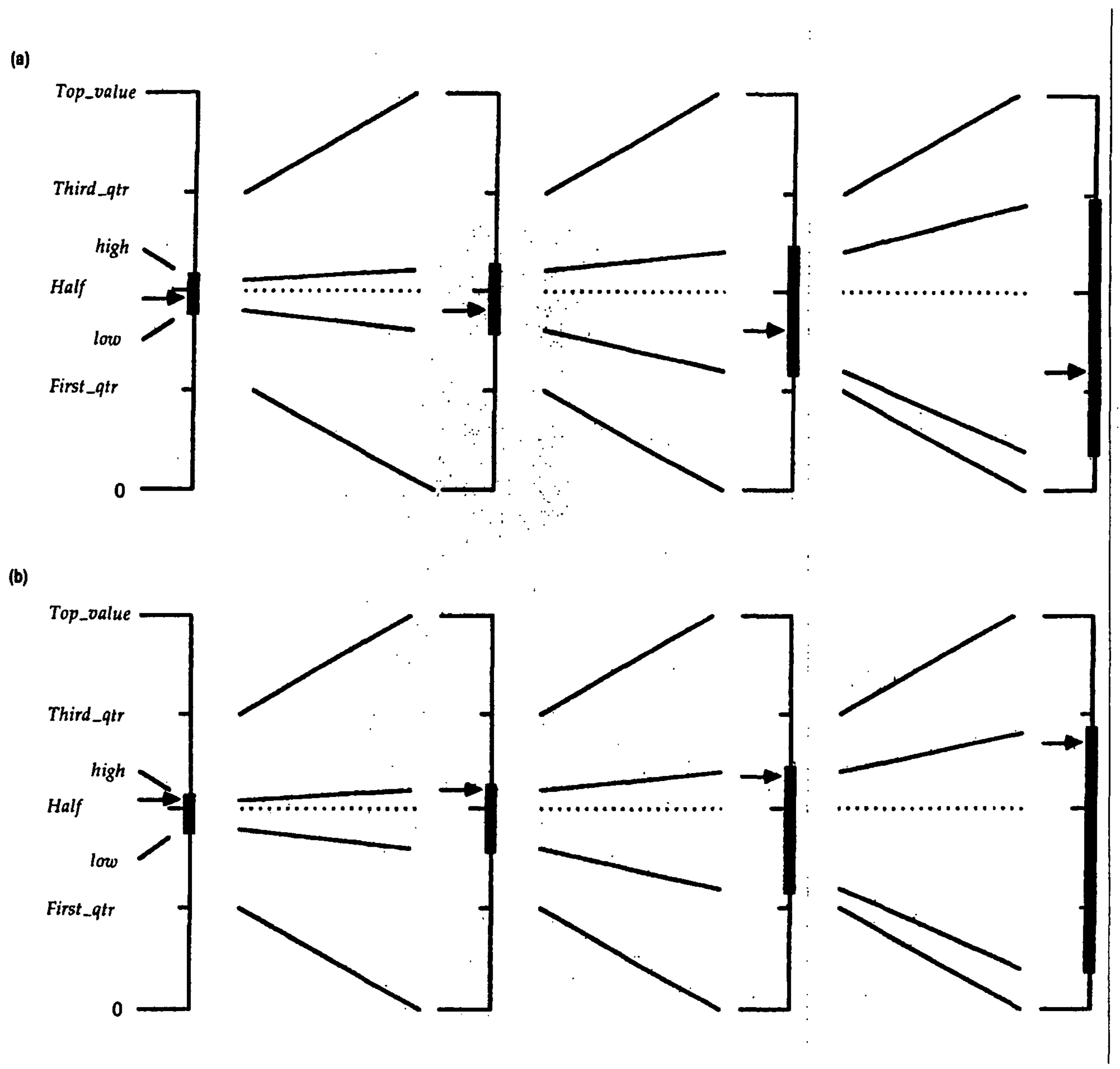

FIGURE 5. Scaling the Interval to Prevent Underflow

of Figure 3. Overflow cannot occur provided the product

range * *Max_frequency*

fits within the integer word length available, since cumulative frequencies cannot exceed *Max_frequency*. *Range* might be as large as *Top_value* + 1, so the largest possible product in Figure 3 is $2^{16}(2^{14} - 1)$, which is less than 2^{30}. *Long* declarations are used for *code_value* (line 7) and *range* (lines 89, 183) to ensure that arithmetic is done to 32-bit precision.

Constraints on the Implementation

The constraints on word length imposed by underflow and overflow can be simplified by assuming that frequency counts are represented in f bits, and *code_values* in c bits. The implementation will work correctly provided

$$f \le c - 2$$
$$f + c \le p,$$

the precision to which arithmetic is performed.

In most C implementations, $p = 31$ if *long* integers are used, and $p = 32$ if they are *unsigned long*. In Figure 3, $f = 14$ and $c = 16$. With appropriately modified declarations, *unsigned long* arithmetic with $f = 15$ and $c = 17$ could be used. In assembly language, $c = 16$ is a natural choice because it expedites some comparisons and bit manipulations (e.g., those of lines 95–113 and 194–213).

If p is restricted to 16 bits, the best values possible are $c = 9$ and $f = 7$, making it impossible to encode a full alphabet of 256 symbols, as each symbol must have a count of at least one. A smaller alphabet (e.g., the 26 letters, or 4-bit nibbles) could still be handled.

Termination

To finish the transmission, it is necessary to send a unique terminating symbol (*EOF_symbol*, line 56) and then follow it by enough bits to ensure that the encoded string falls within the final range. Since *done_encoding*() (lines 119–123) can be sure that *low* and *high* are constrained by either Eq. (1a) or (1b) above, it need only transmit 01 in the first case or 10 in the second to remove the remaining ambiguity. It is convenient to do this using the *bit_plus_follow*() procedure discussed earlier. The *input_bit*() procedure will actually read a few more bits than were sent by *output_bit*(), as it needs to keep the low end of the buffer full. It does not matter what value these bits have, since EOF is uniquely determined by the last two bits actually transmitted.

MODELS FOR ARITHMETIC CODING

The program in Figure 3 must be used with a model that provides a pair of translation tables *index_to_char*[ ] and *char_to_index*[ ], and a cumulative frequency array *cum_freq*[ ]. The requirements on the latter are that

- $cum\_freq[i-1] \geq cum\_freq[i]$;
- an attempt is never made to encode a symbol i for which $cum\_freq[i-1] = cum\_freq[i]$; and
- $cum\_freq[0] \leq Max\_frequency$.

Provided these conditions are satisfied, the values in the array need bear no relationship to the actual cumulative symbol frequencies in messages. Encoding and decoding will still work correctly, although encodings will occupy less space if the frequencies are accurate. (Recall our successfully encoding *eaii!* according to the model of Table I, which does not actually reflect the frequencies in the message.)

Fixed Models

The simplest kind of model is one in which symbol frequencies are fixed. The first model in Figure 4 has symbol frequencies that approximate those of English (taken from a part of the Brown Corpus [12]). However, bytes that did not occur in that sample have been given frequency counts of one in case they do occur in messages to be encoded (so this model will still work for binary files in which all 256 bytes occur). Frequencies have been normalized to total 8000. The initialization procedure *start_model*() simply computes a cumulative version of these frequencies (lines 48–51), having first initialized the translation tables (lines 44–47). Execution speed would be improved if these tables were used to reorder symbols and frequencies so that the most frequent came first in the *cum_freq*[ ] array. Since the model is fixed, the procedure *update_model*(), which is called from both *encode.c* and *decode.c*, is null.

An *exact* model is one where the symbol frequencies in the message are exactly as prescribed by the model. For example, the fixed model of Figure 4 is close to an exact model for the particular excerpt of the Brown Corpus from which it was taken. To be truly exact, however, symbols that did not occur in the excerpt would be assigned counts of zero, rather than one (sacrificing the capability of transmitting messages containing those symbols). Moreover, the frequency counts would not be scaled to a predetermined cumulative frequency, as they have been in Figure 4. The exact model can be calculated and transmitted before the message is sent. It is shown by Cleary and Witten [3] that, under quite general conditions, this will *not* give better overall compression than adaptive coding (which is described next).

Adaptive Models

An adaptive model represents the changing symbol frequencies seen *so far* in a message. Initially all counts might be the same (reflecting no initial information), but they are updated, as each symbol is seen, to approximate the observed frequencies. Provided both encoder and decoder use the same initial values (e.g., equal counts) and the same updating algorithm, their models will remain in step. The encoder receives the next symbol, encodes it, and updates its model. The decoder identifies it according to its current model and then updates its model.

The second half of Figure 4 shows such an adaptive model. This is the type of model recommended for use with Figure 3, for in practice it will outperform a fixed model in terms of compression efficiency. Initialization is the same as for the fixed model, except that all frequencies are set to one. The procedure *update_model*(*symbol*) is called by both *encode_symbol*() and *decode_symbol*() (Figure 3, lines 54 and 151) after each symbol is processed.

Updating the model is quite expensive because of the need to maintain cumulative totals. In the code of Figure 4, frequency counts, which must be maintained anyway, are used to optimize access by keeping the array in frequency order—an effective kind of self-organizing linear search [9]. *Update_model*() first checks to see if the new model will exceed the cumulative-frequency limit, and if so scales all frequencies down by a factor of two (taking care to ensure that no count scales to zero) and recomputes cumulative values (Figure 4, lines 29–37). Then, if necessary, *update_model*() reorders the symbols to place the current one in its correct rank in the frequency ordering, altering the translation tables to reflect the change. Finally, it increments the appropriate frequency count and adjusts cumulative frequencies accordingly.

PERFORMANCE

Now consider the performance of the algorithm of Figure 3, both in compression efficiency and execution time.

Compression Efficiency

In principle, when a message is coded using arithmetic coding, the number of bits in the encoded string is the same as the entropy of that message with respect to the model used for coding. Three factors cause performance to be worse than this in practice:

(1) message termination overhead;
(2) the use of fixed-length rather than infinite-precision arithmetic; and
(3) scaling of counts so that their total is at most *Max_frequency*.

None of these effects is significant, as we now show. In order to isolate the effect of arithmetic coding, the model will be considered to be exact (as defined above).

Arithmetic coding must send extra bits at the end of each message, causing a message termination overhead. Two bits are needed, sent by *done_encoding*() (Figure 3, lines 119–123), in order to disambiguate the final symbol. In cases where a bit stream must be blocked into 8-bit characters before encoding, it will be necessary to round out to the end of a block. Combining these, an extra 9 bits may be required.

The overhead of using fixed-length arithmetic occurs because remainders are truncated on division. It can be assessed by comparing the algorithm's performance with the figure obtained from a theoretical entropy calculation that derives its frequencies from counts scaled exactly as for coding. It is completely negligible—on the order of 10^{-4} bits/symbol.

The penalty paid by scaling counts is somewhat larger, but still very small. For short messages (less than 2^{14} bytes), no scaling need be done. Even with messages of 10^5–10^6 bytes, the overhead was found experimentally to be less than 0.25 percent of the encoded string.

The adaptive model in Figure 4 scales down all counts whenever the total threatens to exceed *Max_frequency*. This has the effect of weighting recent events more heavily than events from earlier in the message. The statistics thus tend to track changes in the input sequence, which can be very beneficial. (We have encountered cases where limiting counts to 6 or 7 bits gives better results than working to higher precision.) Of course, this depends on the source being modeled. Bentley et al. [2] consider other, more explicit, ways of incorporating a recency effect.

Execution Time

The program in Figure 3 has been written for clarity rather than for execution speed. In fact, with the adaptive model in Figure 4, it takes about 420 μs per input byte on a VAX-11/780 to encode a text file, and about the same for decoding. However, easily avoidable overheads such as procedure calls account for much of this, and some simple optimizations increase speed by a factor of two. The following alterations were made to the C version shown:

(1) The procedures *input_bit*(), *output_bit*(), and *bit_plus_follow*() were converted to macros to eliminate procedure-call overhead.
(2) Frequently used quantities were put in register variables.
(3) Multiplies by two were replaced by additions (C "+=").
(4) Array indexing was replaced by pointer manipulation in the loops at line 189 in Figure 3 and lines 49–52 of the adaptive model in Figure 4.

This mildly optimized C implementation has an execution time of 214 μs/252 μs per input byte, for encoding/decoding 100,000 bytes of English text on a VAX-11/780, as shown in Table II. Also given are corresponding figures for the same program on an Apple Macintosh and a SUN-3/75. As can be seen, coding a C source program of the same length took slightly longer in all cases, and a binary object program longer still. The reason for this will be discussed shortly. Two artificial test files were included to allow readers to replicate the results. "Alphabet" consists of enough copies of the 26-letter alphabet to fill out 100,000 characters (ending with a partially completed alphabet). "Skew statistics" contains

TABLE II. Results for Encoding and Decoding 100,000-Byte Files

		VAX-11/780		Macintosh 512 K		SUN-3/75	
	Output (bytes)	Encode time (μs)	Decode time (μs)	Encode time (μs)	Decode time (μs)	Encode time (μs)	Decode time (μs)
Mildly optimized C implementation							
Text file	57,718	214	262	687	881	98	121
C program	62,991	230	288	729	950	105	131
VAX object program	73,501	313	406	950	1,334	145	190
Alphabet	59,292	223	277	719	942	105	130
Skew statistics	12,092	143	170	507	645	70	85
Carefully optimized assembly-language implementation							
Text file	57,718	104	135	194	243	46	58
C program	62,991	109	151	208	266	51	65
VAX object program	73,501	158	241	280	402	75	107
Alphabet	59,292	105	145	204	264	51	65
Skew statistics	12,092	63	81	126	160	28	36

Notes: Times are measured in microseconds per byte of uncompressed data.
The VAX-11/780 had a floating-point accelerator, which reduces integer multiply and divide times.
The Macintosh uses an 8-MHz MC68000 with some memory wait states.
The SUN-3/75 uses a 16.67-MHz MC68020.
All times exclude I/O and operating-system overhead in support of I/O. VAX and SUN figures give user time from the UNIX® *time* command; on the Macintosh, I/O was explicitly directed to an array.
The 4.2BSD C compiler was used for VAX and SUN; Aztec C 1.06g for Macintosh.

10,000 copies of the string *aaaabaaaac*; it demonstrates that files may be encoded into less than one bit per character (output size of 12,092 bytes = 96,736 bits). All results quoted used the adaptive model of Figure 4.

A further factor of two can be gained by reprogramming in assembly language. A carefully optimized version of Figures 3 and 4 (adaptive model) was written in both VAX and M68000 assembly languages. Full use was made of registers, and advantage taken of the 16-bit *code_value* to expedite some crucial comparisons and make subtractions of *Half* trivial. The performance of these implementations on the test files is also shown in Table II in order to give the reader some idea of typical execution speeds.

The VAX-11/780 assembly-language timings are broken down in Table III. These figures were obtained with the UNIX profile facility and are accurate only to within perhaps 10 percent. (This mechanism constructs a histogram of program counter values at real-time clock interrupts and suffers from statistical variation as well as some systematic errors.) "Bounds calculation" refers to the initial parts of *encode_symbol*() and *decode_symbol*() (Figure 3, lines 90–94 and 190–193), which contain multiply and divide operations. "Bit shifting" is the major loop in both the encode and decode routines (lines 95–113 and 194–213). The *cum* calculation in *decode_symbol*(), which requires a multiply/divide, and the following loop to identify the next symbol (lines 187–189), is "Symbol decode." Finally, "Model update" refers to the adaptive *update_model*() procedure of Figure 4 (lines 26–53).

TABLE III. Breakdown of Timings for the VAX-11/780 Assembly-Language Version

	Encode time (μs)	Decode time (μs)
Text file		
Bounds calculation	32	31
Bit shifting	39	30
Model update	29	29
Symbol decode	—	45
Other	4	0
	104	135
C program		
Bounds calculation	30	28
Bit shifting	42	35
Model update	33	36
Symbol decode	—	51
Other	4	1
	109	151
VAX object program		
Bounds calculation	34	31
Bit shifting	46	40
Model update	75	75
Symbol decode	—	94
Other	3	1
	158	241

As expected, the bounds calculation and model update take the same time for both encoding and decoding, within experimental error. Bit shifting was quicker for the text file than for the C program and object file because compression performance was better. The extra time for decoding over encoding is due entirely to the symbol decode step. This takes longer in the C program and object file tests because the loop of line 189 was executed more often (on average 9 times, 13 times, and 35 times, respectively). This also affects the model update time because it is the number of cumulative counts that must be incremented in Figure 4, lines 49–52. In the worst case, when the symbol frequencies are uniformly distributed, these loops are executed an average of 128 times. Worst-case performance would be improved by using a more complex tree representation for frequencies, but this would likely be slower for text files.

SOME APPLICATIONS

Applications of arithmetic coding are legion. By liberating *coding* with respect to a model from the *modeling* required for prediction, it encourages a whole new view of data compression [19]. This separation of function costs nothing in compression performance, since arithmetic coding is (practically) optimal with respect to the entropy of the model. Here we intend to do no more than suggest the scope of this view by briefly considering

(1) adaptive text compression,
(2) nonadaptive coding,
(3) compressing black/white images, and
(4) coding arbitrarily distributed integers.

Of course, as noted earlier, greater coding efficiencies could easily be achieved with more sophisticated models. Modeling, however, is an extensive topic in its own right and is beyond the scope of this article.

Adaptive text compression using single-character adaptive frequencies shows off arithmetic coding to good effect. The results obtained using the program in Figures 3 and 4 vary from 4.8–5.3 bits/character for short English text files (10^3–10^4 bytes) to 4.5–4.7 bits/character for long ones (10^5–10^6 bytes). Although adaptive Huffman techniques do exist (e.g., [5, 7]), they lack the conceptual simplicity of arithmetic coding. Although competitive in compression efficiency for many files, they are slower. For example, Table IV compares the performance of the mildly optimized C implementation of arithmetic coding with that of the UNIX *compact* program that implements adaptive Huffman coding using a similar model. (*Compact*'s model is essentially the same for long files, like those of Table IV, but is better for short files than the model used as an example in this article.) Casual examination of *compact* indicates that the care taken in optimization is roughly comparable for both systems, yet arithmetic coding halves execution time. Compression performance is somewhat better with arithmetic coding on all the example files. The difference would be accentuated with more sophisticated models that predict symbols with probabilities approaching one under certain circumstances (e.g., the letter *u* following *q*).

Nonadaptive coding can be performed arithmetically using fixed, prespecified models like that in the first part of Figure 4. Compression performance will be better than Huffman coding. In order to minimize execution time, the total frequency count, *cum_freq*[0], should be chosen as a power of two so the divisions in the bounds calculations (Figure 3, lines 91–94 and 190–193) can be done as shifts. Encode/decode times of around 60 μs/90 μs should then be possible for an assembly-language implementation on a VAX-11/780. A carefully written implementation of Huffman coding, using table lookup for encoding and decoding, would be a bit faster in this application.

Compressing black/white images using arithmetic coding has been investigated by Langdon and Rissanen [14], who achieved excellent results using

TABLE IV. Comparison of Arithmetic and Adaptive Huffman Coding

	Arithmetic coding			Adaptive Huffman coding		
	Output (bytes)	Encode time (μs)	Decode time (μs)	Output (bytes)	Encode time (μs)	Decode time (μs)
Text file	57,718	214	262	57,781	550	414
C program	62,991	230	288	63,731	596	441
VAX object program	73,546	313	406	76,950	822	606
Alphabet	59,292	223	277	60,127	598	411
Skew statistics	12,092	143	170	16,257	215	132

Notes: The mildly optimized C implementation was used for arithmetic coding.
UNIX *compact* was used for adaptive Huffman coding.
Times are for a VAX-11/780 and exclude I/O and operating-system overhead in support of I/O.

a model that conditioned the probability of a pixel's being black on a template of pixels surrounding it. The template contained a total of 10 pixels, selected from those above and to the left of the current one so that they precede it in the raster scan. This creates 1024 different possible contexts, and for each the probability of the pixel being black was estimated adaptively as the picture was transmitted. Each pixel's polarity was then coded arithmetically according to this probability. A 20–30 percent improvement in compression was attained over earlier methods. To increase coding speed, Langdon and Rissanen used an approximate method of arithmetic coding that avoided multiplication by representing probabilities as integer powers of ½. Huffman coding cannot be directly used in this application, as it never compresses with a two-symbol alphabet. Run-length coding, a popular method for use with two-valued alphabets, provides another opportunity for arithmetic coding. The model reduces the data to a sequence of lengths of runs of the same symbol (e.g., for picture coding, run-lengths of black followed by white followed by black followed by white . . .). The sequence of lengths must be transmitted. The CCITT facsimile coding standard [11] bases a Huffman code on the frequencies with which black and white runs of different lengths occur in sample documents. A fixed arithmetic code using these same frequencies would give better performance; adapting the frequencies to each particular document would be better still.

Coding arbitrarily distributed integers is often called for in use with more sophisticated models of text, image, or other data. Consider, for instance, the locally adaptive data compression scheme of Bentley et al. [2], in which the encoder and decoder cache the last N different words seen. A word present in the cache is transmitted by sending the integer cache index. Words not in the cache are transmitted by sending a new-word marker followed by the characters of the word. This is an excellent model for text in which words are used frequently over short intervals and then fall into long periods of disuse. Their paper discusses several variable-length codings for the integers used as cache indexes. Arithmetic coding allows *any* probability distribution to be used as the basis for a variable-length encoding, including—among countless others—the ones implied by the particular codes discussed there. It also permits use of an adaptive model for cache indexes, which is desirable if the distribution of cache hits is difficult to predict in advance. Furthermore, with arithmetic coding, the code spaced allotted to the cache indexes can be scaled down to accommodate any desired probability for the new-word marker.

APPENDIX. Proof of Decoding Inequality

Using one-letter abbreviations for *cum_freq*, *symbol*, *low*, *high*, and *value*, suppose

$$c[s] \le \left\lfloor \frac{(v-l+1)\times c[0]-1}{h-l+1} \right\rfloor < c[s-1];$$

in other words,

$$\begin{aligned} c[s] &\le \frac{(v-l+1)\times c[0]-1}{r} - \varepsilon \\ &\le c[s-1]-1, \end{aligned} \qquad (1)$$

where

$$r = h - l + 1, \qquad 0 \le \varepsilon \le \frac{r-1}{r}.$$

(The last inequality of Eq. (1) derives from the fact that $c[s-1]$ must be an integer.) Then we wish to show that $l' \le v \le h'$, where l' and h' are the updated values for *low* and *high* as defined below.

(a) $$\begin{aligned} l' &\equiv l + \left\lfloor \frac{r \times c[s]}{c[0]} \right\rfloor \\ &\le l + \frac{r}{c[0]}\left[\frac{(v-l+1)\times c[0]-1}{r} - \varepsilon\right] \qquad \text{from Eq. (1),} \\ &\le v + 1 - \frac{1}{c[0]}, \end{aligned}$$

so $l' \le v$ since both v and l' are integers and $c[0] > 0$.

(b) $$\begin{aligned} h' &\equiv l + \left\lfloor \frac{r \times c[s-1]}{c[0]} \right\rfloor - 1 \\ &\ge l + \frac{r}{c[0]}\left[\frac{(v-l+1)\times c[0]-1}{r} + 1 - \varepsilon\right] - 1 \qquad \text{from Eq. (1),} \\ &\ge v + \frac{r}{c[0]}\left[-\frac{1}{r} + 1 - \frac{r-1}{r}\right] = v. \end{aligned}$$

REFERENCES

1. Abramson, N. *Information Theory and Coding.* McGraw-Hill, New York, 1963. This textbook contains the first reference to what was to become the method of arithmetic coding (pp. 61–62).
2. Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr. 1986), 320–330. Shows how recency effects can be incorporated explicitly into a text compression system.

3. Cleary, J.G., and Witten, I.H. A comparison of enumerative and adaptive codes. *IEEE Trans. Inf. Theory IT-30*, 2 (Mar. 1984), 306–315. Demonstrates under quite general conditions that adaptive coding outperforms the method of calculating and transmitting an exact model of the message first.
4. Cleary, J.G., and Witten, I.H. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun. COM-32*, 4 (Apr. 1984), 396–402. Presents an adaptive modeling method that reduces a large sample of mixed-case English text to around 2.2 bits/character when arithmetically coded.
5. Cormack, G.V., and Horspool, R.N. Algorithms for adaptive Huffman codes. *Inf. Process. Lett. 18*, 3 (Mar. 1984), 159–166. Describes how adaptive Huffman coding can be implemented efficiently.
6. Cormack, G.V., and Horspool, R.N. Data compression using dynamic Markov modeling. Res. Rep., Computer Science Dept., Univ. of Waterloo, Ontario, Apr. 1985. Also to be published in *Comput. J.* Presents an adaptive state-modeling technique that, in conjunction with arithmetic coding, produces results competitive with those of [4].
7. Gallager, R.G. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory IT-24*, 6 (Nov. 1978), 668–674. Presents an adaptive Huffman coding algorithm, and derives new bounds on the redundancy of Huffman codes.
8. Held, G. *Data Compression: Techniques and Applications.* Wiley, New York, 1984. Explains a number of ad hoc techniques for compressing text.
9. Hester, J.H., and Hirschberg, D.S. Self-organizing linear search. *ACM Comput. Surv. 17*, 3 (Sept. 1985), 295–311. A general analysis of the technique used in the present article to expedite access to an array of dynamically changing frequency counts.
10. Huffman, D.A. A method for the construction of minimum-redundancy codes. *Proc. Inst. Electr. Radio Eng. 40*, 9 (Sept. 1952), 1098–1101. The classic paper in which Huffman introduced his famous coding method.
11. Hunter, R., and Robinson, A.H. International digital facsimile coding standards. *Proc. Inst. Electr. Electron. Eng. 68*, 7 (July 1980), 854–867. Describes the use of Huffman coding to compress run lengths in black/white images.
12. Kucera, H., and Francis, W.N. *Computational Analysis of Present-Day American English.* Brown University Press, Providence, R.I., 1967. This large corpus of English is often used for computer experiments with text, including some of the evaluation reported in the present article.
13. Langdon, G.G. An introduction to arithmetic coding. *IBM J. Res. Dev. 28*, 2 (Mar. 1984), 135–149. Introduction to arithmetic coding from the point of view of hardware implementation.
14. Langdon, G.G., and Rissanen, J. Compression of black-white images with arithmetic coding. *IEEE Trans. Commun. COM 29*, 6 (June 1981), 858–867. Uses a modeling method specially tailored to black/white pictures, in conjunction with arithmetic coding, to achieve excellent compression results.
15. Pasco, R. Source coding algorithms for fast data compression. Ph.D. thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif., 1976. An early exposition of the idea of arithmetic coding, but lacking the idea of incremental operation.
16. Rissanen, J.J. Generalized Kraft inequality and arithmetic coding. *IBM J. Res. Dev. 20* (May 1976), 198–203. Another early exposition of the idea of arithmetic coding.
17. Rissanen, J.J. Arithmetic codings as number representations. *Acta Polytech. Scand. Math. 31* (Dec. 1979), 44–51. Further develops arithmetic coding as a practical technique for data representation.
18. Rissanen, J., and Langdon, G.G. Arithmetic coding. *IBM J. Res. Dev. 23*, 2 (Mar. 1979), 149–162. Describes a broad class of arithmetic codes.
19. Rissanen, J., and Langdon, G.G. Universal modeling and coding. *IEEE Trans. Inf. Theory IT-27*, 1 (Jan. 1981), 12–23. Shows how data compression can be separated into *modeling* for prediction and *coding* with respect to a model.
20. Rubin, F. Arithmetic stream coding using fixed precision registers. *IEEE Trans. Inf. Theory IT-25*, 6 (Nov. 1979), 672–675. One of the first papers to present all the essential elements of practical arithmetic coding, including fixed-point computation and incremental operation.
21. Shannon, C.E., and Weaver, W. *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, Ill., 1949. A classic book that develops communication theory from the ground up.
22. Welch, T.A. A technique for high-performance data compression. *Computer 17*, 6 (June 1984), 8–19. A very fast coding technique based on the method of [23], but whose compression performance is poor by the standards of [4] and [6]. An improved implementation of this method is widely used in UNIX systems under the name *compress.*
23. Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory IT-24*, 5 (Sept. 1978), 530–536. Describes a method of text compression that works by replacing a substring with a pointer to an earlier occurrence of the same substring. Although it performs quite well, it does not provide a clear separation between modeling and coding.

CR Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*; H.1.1 [**Models and Principles**]: Systems and Information Theory—*information theory*
General Terms: Algorithms, Performance
Additional Key Words and Phrases: Adaptive modeling, arithmetic coding, Huffman coding

Received 8/86; accepted 12/86

Authors' Present Address: Ian H. Witten, Radford M. Neal, and John G. Cleary, Dept. of Computer Science, The University of Calgary, 2500 University Drive NW, Calgary, Canada T2N 1N4.